THE METACOMCO TOOLKIT

Software for the AMIGA

METACOMCO

THE METACOMCO TOOLKIT

Software for the AMIGA

METACOMCO

COPYRIGHT

The manual *The Metacomco Toolkit* Copyright (c) 1987 Metacomco plc. All Rights Reserved. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from Metacomco plc.

Metacomco Toolkit software Copyright (c) 1987, Metacomco plc. All Rights Reserved. The distribution and sale of this product are intended for the use of the original purchaser only. Lawful users of this program are hereby licensed only to read the program, from its medium into memory of a computer, solely for the purpose of executing the program. Duplicating, copying, selling, or otherwise distributing this product is a violation of the law.

DISCLAIMER

THIS PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT THE DEVELOPER OR METACOMCO PLC OR ITS AFFILIATED DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FURTHER, METACOMCO PLC OR ITS AFFILIATED COMPANIES DO NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK. IN NO EVENT WILL METACOMCO PLC OR ITS AFFILIATED COMPANIES BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAM EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

Amiga, AmigaDOS, and Workbench are trademarks of
Commodore-Amiga, Inc.

This manual refers to Release 1.2, June 1987.

Printed in the U.K.

Table of Contents

Part 1: Introduction

Introducing the Amiga Toolkit	Page 1
Making a Backup Copy	Page 1
Using this Manual	Page 4
Format Conventions	Page 4
Template Conventions	Page 6
Quick Reference List	Page 8

Part 2: Commands and Utilities

ALIB	Page 11
AUX-CLI	Page 13
BROWSE	Page 15
DISASM	Page 16
ENLARGE	Page 17
MAKE	Page 18
MOUNT	Page 45
PACK	Page 46
PIPES	Page 47
TOUCH	Page 48
UNPACK	Page 49

Customer Support

1.1 Introducing the Amiga Toolkit

This manual is a user reference manual for the Amiga Toolkit. The Amiga Toolkit is a collection of additional AmigaDOS commands and utilities written by Metacomco, the creators of AmigaDOS. These commands and utilities allow you to make better use of the power of the AmigaDOS Command Line Interface (CLI).

A full description of the AmigaDOS CLI can be found in the AmigaDOS manuals, which can be obtained from Commodore-Amiga Inc. or as a single volume from Bantam Books.

Before you start to use the Amiga Toolkit, you should ensure that you have the following items:

1. Kickstart disk
2. CLI bootable disk
3. Amiga Toolkit disk

In this manual, all references to a 'disk' refer to a 3.5 inch diskette.

1.2 Making a Backup Copy

It is important that you make a backup copy of the Amiga Toolkit disk and keep the master copy in a safe place. You should do this before you even start to use any of the commands or utilities on the disk.

Now you should merge the utilities on the Toolkit disk with those on your CLI disk. To do so you must follow one of the procedures outlined below.

Two Drive System

If you have two (or more) drives, place your CLI disk in drive 0 (the internal drive, df0:) and the Toolkit disk in drive 1 (the external drive, df1:). Now type the following commands:

```
MAKEDIR sys:make
ASSIGN make sys:make
COPY df1:c TO c:
COPY df1:l TO l:
COPY df1:make to make:
```

(Note that :l is lower case :L; it is not :1.) Once you've copied the C and L directories you should modify the system file DEVS:MOUNTLIST (see below for instructions).

One Drive System

For a one drive system, you should make use of the RAM device, often known as the ramdisk, to perform the copy. Copy the files from the Toolkit disk to the ramdisk, then copy the information from the ramdisk to your CLI disk. Your CLI disk should be in the internal disk drive before you issue the following command:

```
MAKEDIR sys:make
ASSIGN make sys:make
COPY Toolkit: to ram: all
```

At this point the system will put up a requestor for the Toolkit disk so insert it.

The system may then ask you to put the CLI disk back in again so that it can load the ramdisk handler from l:, in which case you should do so. Of course, it will then ask you for the Toolkit disk again, once it has loaded the ramdisk handler from l:.

Once the system has finished copying from the Toolkit disk, type

```
COPY ram:c to c:
```

At this point the system will put up a requestor for your original CLI disk and so you should replace it in the drive.

Finally, type

```
COPY ram:l to l:
COPY ram:make to make:
```

Once the files have been copied, you have to modify the system file DEVS:MOUNTLIST.

Modifying the DEVS:MOUNTLIST File

This file contains information about devices which are not an integral part of the system. Fo example, if you have a hard disk, the file DEVS:MOUNTLIST will contain information describing how to integrate the hard disk into the base system.

DEVS:MOUNTLIST is a text file which contains information to be processed by the MOUNT command.

Since the Toolkit disk provides you with two new devices, pipe: and aux:, you must modify the DEVS:MOUNTLIST file to give the system information about these two devices.

Type

```
ED DEVS:MOUNTLIST
```

Go to the bottom of the file and add the following lines

```
AUX:        Handler = L:aux-handler
            Stacksize = 700
            Priority = 5
#
```

```
PIPE:     Handler = L:pipe-handler
          Stacksize = 700
          Priority = 5
#
```

This adds two new entries in the mountlist, called AUX: and PIPE:.

A sample MOUNTLIST file is supplied on the Toolkit disk in the DEVS directory, and this file contains entries for AUX: and PIPE:, so you should consult this file if you are in any doubt.

Preparing to use the MAKE utility

The Make utility which is included in this toolkit uses the creation date and time of files. It is vital for the correct running of Make that the system clock is set to the correct date and time. The CLI command DATE is used to set the date and time. It is suggested that you include a call to DATE in your startup sequence. To do this, edit the file s:startup-sequence and add the line:

```
DATE ?
```

Make requires T: to be assigned. This is done by inserting the following line in your startup file.

```
assign t: sys:t
```

1.3 Using this Manual

The last part of this manual contains a specification of each of the commands and utilities provided in the Toolkit. The layout and style has been chosen to match that of the AmigaDOS manuals. As some readers may be unfamiliar with this format, here is a quick reminder:

Format: a symbolic representation of what you can type when you give the command. A description of the symbols can be found below under **Format Conventions**.

Template:	a full template of the command. A description of templates can be found below under **Template Conventions**.
Purpose:	a brief description of what the command or utility does.
Specification:	a full description of the command or utility.
Examples:	examples of the command or utility's use.
See also:	a list of other relevant commands.

1.3.1 Format Conventions

The conventions used in this manual are in common use. However, as there is no standard, some explanation of these conventions is needed. The following list should help to overcome any difficulties you may have in understanding the format explanations.

Symbol	Action
< >	used to enclose what is expected at that position. For example, BROWSE <filename> means that you can type something like BROWSE df0:myfile (without the enclosing < >). You do NOT type the text "<filename>."
[ ]	used to enclose anything that is optional. For example, combining the above convention, [<filename>] means that you may give a filename at that position, but you do not have to do so. These square brackets can enclose one or many items and can be nested. Anything enclosed in nested square brackets may be given if whatever is enclosed in the outer brackets is given.
|	used to separate mutually exclusive alternatives (either or). For example, A|ADD means you can type either A or ADD.

... used to denote a continuing series. For example, a,b,c,...z means a continuing series of letters from a to z.

Anything else that appears in a format explanation outside the angle or square brackets should be treated as literal. For example, BROWSE <filename> means that you type BROWSE and then give the filename of the file you wish to browse through.

Examples of what you type

```
appear like this.
```

Examples of what the system displays

```
appear like this.
```

1.3.2 Template Conventions

An argument template is included in the documentation of each of the commands. An argument template is the expected pattern of a particular command; it specifies a list of keywords that you may use and the order in which they are expected. The keywords are enclosed in double quotes and, if there is more than one, they are separated by commas. You do not type the quotes or the commas; they are just there to help separate the items in the list. Where there are synonyms, the alternatives appear in the template separated by an equals sign (=). You can use either form, but don't give both. That is, for A=ADD, give A or ADD, but not A=ADD.

The keywords specify the number and form of the arguments that the program expects. The arguments may be optional or required. If you give the arguments, you may specify them in one of two ways:

By position In this case, you provide the arguments in the same order as the keyword list indicates.

By keyword In this case, the order does not matter, and you precede each argument with the relevant keyword.

The keywords in these argument lists have certain qualifiers associated with them. These qualifiers are represented by a slash (/) and a specific letter. The meanings of the qualifiers are as follows:

/A	The argument is required and may not be omitted.
/K	The argument must be given with the keyword and may not be used positionally.
/S	The keyword is a switch (that is, a toggle) and takes no argument.

The qualifiers A and K may be combined; in which case you must give the argument and keyword.

If the arguments you specify do not match the template, most commands simply display the message 'Bad args' or 'Bad arguments' and stop. You must retype the command name and argument. If you need to be reminded of a command's template, follow these steps:

1. Type the command name
2. Press the space bar
3. Type ?
4. Press RETURN

For example, if you want to find out what arguments the BROWSE command expects, you type

```
BROWSE ?
```

AmigaDOS then returns

```
FILE/A:
```

and waits. You can then type the filename and press RETURN to execute BROWSE. (Press CTRL-C and then RETURN if you do not wish to execute BROWSE.)

1.4 Quick Reference List

This section provides a quick reference to the commands and utilities described in the reference part of this manual.

Commands

Name	Format, Template, and Purpose
ALIB	ALIB <lib> <filename> [AS|NAME <name>] [HUNK <hunkname>] [XDEF <symbolname>] [[A|ADD] |[D|DEL|DELETE] |[R|REPLACE]|[X|EXTRACT] |[DIR]] [N|NOCHECKS] ALIB "LIBRARY/A,FILE,AS=NAME/K,HUNK/K, XDEF/K,A=ADD/S,D=DEL=DELETE/S, R=REPLACE/S,X=EXTRACT/S,DIR/S, N=NOCHECKS/S" Creates or amends an Amiga library module
BROWSE	BROWSE <filename> BROWSE "FILE/A" Displays the contents of a file

Name	Format, Template, and Purpose
DISASM	DISASM [FROM] <filename> [[TO] <filename>] [OPT <opt>] DISASM "FROM/A,TO,OPT/K" Disassembles an object module
ENLARGE	ENLARGE <string> [TO <filename>] [FW | FIELDWIDTH <n>] ENLARGE "/A,TO/K,FW=FIELDWIDTH/K" Prints a string in large letters
MAKE	MAKE [[TARGET] <filename>][-F <filename>] [-T][-N][-S] MAKE "TARGET,-F,-T/S,-N/S,-S/S" Creates latest version of a target file.
MOUNT	MOUNT <device> MOUNT "DEV/A" Mounts a new device
PACK	PACK [FROM] <filename> [TO] <filename> [[WORKSIZE | WS] <n>] PACK "FROM/A,TO/A,WORKSIZE=WS" Translates a text file to a compact form

TOUCH TOUCH [FILE] <filename>

TOUCH "FILE/A"

Updates a file's creation date

UNPACK UNPACK [FROM] <filename> [TO] <filename> [[WORKSIZE | WS] <n>]

UNPACK "FROM/A,TO/A,WORKSIZE=WS"

Regenerates a text file from a compacted form

Utilities

Name **Format**

PIPES PIPE:[<string>]

Communicates I/O between tasks

AUX-CLI AUX:

Console handler for the serial line

ALIB

Format:

```
ALIB <lib> <filename> [AS|NAME <name>]
[HUNK <hunkname>]
[XDEF <symbolname>]
[[A|ADD]
|[D|DEL|DELETE]
|[R|REPLACE]
|[X|EXTRACT]
|[DIR]]
[N|NOCHECKS]
```

Template:

```
ALIB "LIBRARY/A,FILE,AS=NAME/K,
 HUNK/K,XDEF/K,A=ADD/S,
 D=DEL=DELETE/S,R=REPLACE/S,
 X=EXTRACT/S,DIR/S,N=NOCHECKS/S"
```

Purpose: To create or amend an Amiga library module.

Specification:

The LIBRARY parameter specifies the name of the library to be amended.

The FILE parameter specifies either the name of a file to be added to the library, or the name of the file to be created with an extracted program unit.

The NAME parameter specifies either the name to be given to a program unit being added to the library or the name of a program unit the user wishes to extract, delete, or replace.

The HUNK parameter specifies a hunk name contained in the program unit the user wishes to extract, delete, or replace.

The XDEF parameter specifies a symbol that is defined in the program unit the user wishes to extract, delete, or replace.

Note: Any combination of the three parameters NAME, HUNK, XDEF may be used when extracting, deleting, or replacing. The first program unit in the library to satisfy one of these requests will be the one operated upon.

The ADD, DELETE, REPLACE, EXTRACT, and DIR parameters are switches that you can give to select which action you require. For instance, DIR lists on the screen all the program unit names, hunk names, and symbols within the library. Only one action may be selected at any one time.

The NOCHECKS parameter is also a switch. If you select NOCHECKS, ALIB will not check the validity of LIBRARY when adding a new program unit, but simply appends it to the library. This option greatly increases the speed in which ALIB adds modules.

Examples:

```
ALIB library mylib file module1 add
```

adds the file 'module1' to the library 'mylib. If mylib does not exist, it is created.

```
ALIB clib name printf delete
```

deletes the program unit 'printf' from the library 'clib'.

```
ALIB clib extract module xdef _fopen
```

extracts the program unit containing the external definition for the symbol '__fopen' placing it in the file 'module.'

AUX-CLI

Format: AUX:

Purpose: A console handler for the serial-line.

Specification:

The Auxiliary Command Line Interface (Aux-CLI) is a console handler with the device name aux:. Aux: is similar to the device con:, except that it handles input and output to the serial line port rather than the window system.

The action of aux: is similar to that of con:. For example, it handles the following special keys in exactly the same way:

BACKSPACE	Deletes a character
CTRL-X	Deletes a whole line
CTRL-S	Stops output
CTRL-Q	Resumes output

If you use con: and you type anything in, AmigaDOS temporarily suspends all output to con:. In other words, AmigaDOS politely waits until you have completed a line before sending any messages to you. This means that not only are you less likely to miss anything, you are less likely to be confused. Exactly the same thing happens when you use aux:; if you type anything, AmigaDOS suspends output to aux:.

Aux: has many uses. It is useful for remote debugging, particularly of large graphics programs. This is because you can display a full screen of debugging information without disturbing the main Amiga screen. Its main use, however, is to let you attach another terminal to your Amiga's serial line port. This terminal can then be used by a second user using another Command Line Interface (CLI).

In order to use aux: you must first make the device available by mounting it with MOUNT (see MOUNT later in this manual for further details on mounting devices). Next, you must use the Workbench

"Preferences" to set up the particular characteristics of the port that you require, such as the baud rate. Now you can use aux: as an argument to a CLI command, as in the example below (NEWCLI aux:), or you can use aux: in a program by calling Open("AUX:").

There are a couple of points to remember:

- Programs or commands that open windows will still work under aux: but you'll find that they open windows on the Amiga's screen, and not on the remote terminal.

- Nothing else must use the serial line while you use aux:.

Examples:

```
MOUNT aux:
NEWCLI aux:
```

sets up an extra CLI reading and writing from the serial port.

BROWSE

Format: BROWSE <filename>

Template: BROWSE "FILE/A"

Purpose: To view the contents of a file.

Specification:

The FILE parameter specifies the file to be browsed. BROWSE displays the contents of this file in the output window that it creates. The contents of the window will not scroll up unless you explicitly request it to do so. When BROWSE has displayed a windowful of text, it gives the prompt 'MORE ?' at the bottom of the window. You can then give one of of the following commands:

<space>	Display the next windowful of text.
<return>	Diplay the next line of text.
s	Skip the next windowful of text.
f	Skip the next line of text.
=	Display the current line number.
r	Rewind to the start of the file.
\<string>	Do a case sensitive search for the next occurrence of the text <string>.
/<string>	Do a non-case sensitive search for the next occurrence of the text <string>.
n	Find the next occurrence of the previously defined text <string>.
h	Display a help message.
q	Quit from BROWSE.

Examples:

```
BROWSE myfile.text
```

allows you to browse through the file 'myfile.text'.

DISASM

Format: DISASM [FROM] <filename> [[TO] <filename>] [OPT <opt>]

Template: DISASM "FROM/A,TO,OPT/K"

Purpose: To disassemble an object module.

Specification:

The FROM parameter specifies the file to be disassembled. The TO parameter specifies an output file; otherwise output goes to the terminal.

Options:

X Produce an output that will be accepted by an assembler.
N Disassemble a non-Amiga format module.
Wn Set workspace size (in bytes).

The default workspace size is 20000 bytes.

Examples:

```
DISASM myfile.o to myfile.dis opt x
```

disassembles the file 'myfile.o' to the file 'myfile.dis'. The output will be in a form that is accepted by the Macro Assembler.

The disassembler assumes that it is disassembling an object module produced by an Amiga compiler, assembler, or linker unless you use the N option.

ENLARGE

Format: ENLARGE <string> [TO <filename>]
[FW|FIELDWIDTH <n>]

Template: ENLARGE "/A,TO/K,FW=FIELDWIDTH/K"

Purpose: To print a string in large letters.

Specification:

The first argument is the string to be enlarged. If necessary, ENLARGE truncates the string so that it will fit in the field width you specified with the last argument. It also centers the string within the specified field width. However, if you don't specify a field width, ENLARGE left-justifies the entire string. The width of the field is measured in standard-size character positions (columns). The TO parameter specifies an output file; otherwise the output goes to the terminal.

Examples:

```
ENLARGE Hello!
```

displays a large version of the text 'Hello!' on the screen.

```
ENLARGE "4TH JULY" TO signpost FW 80
```

sends the string "4TH JULY" to the file 'signpost' and centers it in a field width of 80 columns.

MAKE

Format: MAKE [[TARGET] <filename>][-F <filename>] [-T][-N][-S]

Template: MAKE "TARGET,-F,-T/S,-N/S,-S/S"

Purpose: Creates latest version of a target file.

Specification:

Since Make is a more complex utility than the others in this toolkit, its description is divided into seven sections:

- Introducing Make
- A Closer Look at Make
- Makefiles
- Executing Commands
- Invoking Make
- Differences from UNIX Make
- Makefile Examples

Introducing Make

A computing project, such as developing an applications program or producing documentation, involves the processing of files (for example compilation, linking, text processing). Keeping track of the files involved, and deciding which file or files should undergo which process, is a time consuming task. It is a task that can easily be automated.

Make is a program that maintains the files created from other files in a computing project. The primary function of Make is to ensure that every such file is up-to-date. If Make finds a file that is not up-to-date then it will execute the commands necessary to recreate that file, after first ensuring that all the files required to recreate that file are also up-to-date.

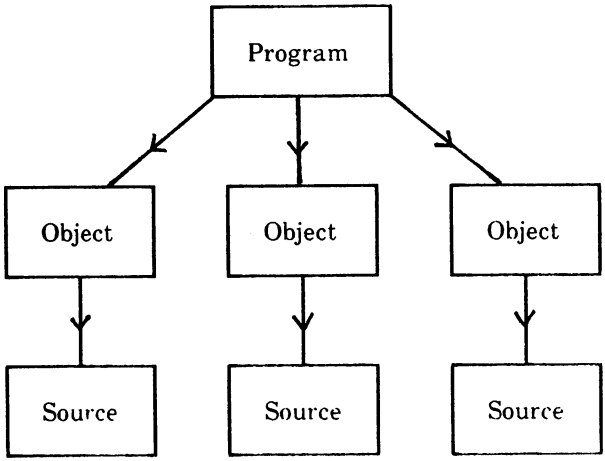
Make is really a clever batch program. A standard batch file simply contains a list of commands to invoke, whereas Make's batch file, called a makefile, contains rules for deciding which commands to invoke.

These rules use the idea of file dependencies. The usual example of file dependency is between a source file and the object file resulting from compilation of the source file. The object file depends on the source file. If you change the source file, the object is no longer up to date and the source file must be recompiled in order to recreate the object file. More formally file 'A' depends on file 'B' means that when file 'B' is updated, either by editing or by some other process, it is necessary to recreate file 'A'. It is possible for one file to depend on several other files. For instance, a program file produced by a linker depends on all the object files that must be linked together to produce it. Our original example of source file and object file may also involve other dependencies. There may, for example, be one or more header files included in the source. In this case the object file also depends on each of these header files.

Throughout this chapter two terms connected with file dependencies are used. These are the terms 'target' and 'prerequisite'. If file 'A' depends on file 'B' then file 'A' is the target and file 'B' is a prerequisite file of file 'A'.

Of course individual file dependencies vary according to the project you are working on. This is where a makefile comes in. It specifies the dependencies between the files used in a particular project. It also specifies the command sequences that must be invoked to recreate each file. You will need a makefile associated with each project.

The diagram below illustrates file dependency. Each box represents a file. An arrow leaving a box means that the file depends on the file the arrow is going to. Thus the program file depends on all the object files, and each object file depends on its source file. This means that all the object files are prerequisites of the target program file.



A makefile is essentially a list of rules which govern under what circumstances each file is to be recreated and how this is to be done. There are two varieties of rule which the makefile may contain. The first is the explicit rule. Explicit rules name each file involved in any dependency explicitly along with a command sequence to recreate the target. The second variety of rule is the implicit rule. Implicit rules specify general dependencies between files of different types. These two types of rules are described in detail later on.

To reduce typing, Make has a macro facility that substitutes strings for macro names within the makefile. Macros are also described more fully at a later stage in this manual.

A Closer Look at Make

Whenever Make is invoked it must be given the name of the primary target file. The primary target file is the file that Make has to ensure is up-to-date. To achieve this Make will need to ensure that each of its prerequisite files are up-to-date, and that each of those files prerequisite files are also up-to-date, and so on recursively. If Make finds a file that is not up-to-date, it will recreate it using the command sequence specified in the makefile.

This procedure ensures that only those files that need updating are recreated. If the primary target file is up-to-date, then Make will not do anything. In program development the primary target file is the actual program.

The primary target file may either be specified in the parameter line or in the makefile. If specified in the makefile it is taken to be the first target file in the first explicit rule defined in the makefile.

Whenever you wish either to create the latest version of your program or to check that the version you have is the latest, then invoke Make by typing the line

```
make
```

All you have to do is to write a makefile for your project, but this only needs to be done once.

Makefiles

Because the makefile is so essential to the correct operation of Make a large proportion of the remainder of this chapter is devoted to explaining how to write one.

When Make is invoked it first reads in your makefile. By default Make looks for a makefile called 'MAKEFILE'. If you wish to call your makefile something else, you must specify the name of your makefile in the command line, using the -F option.

As already stated, the makefile contains the information that Make requires about your files, that is, their dependencies and the commands that must be invoked to recreate each target from its prerequisite files.

The makefile contains four types of entry: explicit rules, implicit rules, directives and macros. The format and significance of each of these is explained in the following subsections.

Explicit Rules

An explicit rule consists of one or more target files, a list of prerequisite files (possibly none), and, optionally, a command sequence to be executed if one of the targets must be recreated.

The three parts of the entry are separated from each other in the following manner. The list of targets must be all on the same line separated by spaces and be terminated by a colon. The list of prerequisites follow on the same line, again separated by spaces. Any command sequence then follows either on the same line as the targets and prerequisites, separated from them by a semicolon, or on the next line, in which case the line must start with blank space (any combination of spaces and tabs).

```
<targets> : <prerequisites> ; <command sequence>
```

or

```
<target> ; <prerequisites>
  <command sequence>
```

Note that the colon must be preceded by a blank space since filenames may contain colons. If a filename is followed by a colon then a blank space must also separate them. Note that the hash character is used to introduce a comment. You can improve on this makefile as your knowledge of Make's capabilities improves.

A command is made up of two parts; the pathname of a disk-based program and the parameter line to be passed to the program. These are separated by blank space. A command sequence is a number of commands, each command being specified on a separate line.

```
assem myfile.asm to myfile.o
```

Command sequences may go over several lines as long as each supplementary line has leading blanks. Alternatively, the entire command sequence may be enclosed in square brackets; in which case a freer format for the commands is possible, since all text after the opening square bracket until the closing square bracket is assumed to be part of the command sequence. Square brackets may be nested within the delimiting ones. The opening square bracket must be on the same line as the targets and prerequisites, the semicolon then being optional. For example:

```
myfile.o : myfile.asm ; echo "assembling myfile.asm"
   assem myfile.asm to myfile.o
```

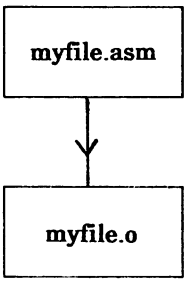
or

```
myfile.o : myfile.asm
   echo "assembling myfile.asm"
   assem myfile.asm to myfile.o
```

or

```
myfile.o : myfile.asm [
   echo "assembling myfile.asm"
   assem myfile.asm to myfile.o ]
```

are all valid ways of writing the same explicit rule, the meaning of which is: the file 'myfile.o' depends on the file 'myfile.asm'. A diagram to display this dependency is as follows:



The written rule holds one piece of extra information and that is the command sequence needed to recreate the target. In this example the commands 'echo "assembling myfile.asm"' and 'assem myfile.asm to myfile.o' are to be executed if 'myfile.o' is not up-to-date; that is, if 'myfile.asm' has been modified since the last time 'myfile.o' was created. Note echo is a program that writes its parameter line to the terminal.

Both the list of prerequisites and the command sequence are optional. If no prerequisite file is present then the command sequence will always be executed provided the primary target file depends on the target file of the rule, either directly or indirectly. If no command sequence is present then the entry just specifies a dependency. The lack of both a prerequisite file and a command sequence is not permitted since such an entry could have no useful purpose.

More than one target may appear in a single entry. This is used to save space in the makefile by reducing several entries to a single one.

A file may appear as a target or prerequisite in more than one explicit rule. If a file appears as a target in more than one explicit rule, only one of those rules can have a command associated with it. The dependencies defined in your makefile must not be cyclic, that is, a file cannot depend on itself, either directly or indirectly.

The following example shows a makefile for developing a program called 'calc'. The program is constructed from some C and Assembler sources.

```
#
# Example makefile - first attempt
#

# entry 1
calc : go.o io.o main.o eval.o calc.o
   alink "go.o io.o main.o eval.o" to calc

# entry 2
go.o : go.asm
    assem go.asm to go.o
# entry 3
io.o  : io.asm
    assem io.asm to io.o
# entry 4
main.o : main.c ;  CC main
# entry 5
eval.o : eval.c ;  CC eval
# entry 6
calc.o : calc.c ;  CC calc

# entry 7
go.o io.o : calc.i
# entry 8
main.o eval.o calc.o : stdio.h calc.h
```

Entry 1 specifies that the primary target 'calc' depends on the five files 'go.o', 'io.o', 'main.o', 'eval.o' and 'calc.o'. If one or more of these five files has a last-modified date later than that of 'calc', then 'calc' should be reconstructed by executing the command

```
alink "go.o io.o main.o eval.o calc.o" to calc
```

This instructs the linker, alink, to link together the five '.o' files to construct 'calc'.

The next five entries (entry 2 to entry 6) in the makefile describe how to make the five '.o' files from their respective sources. For example, the file 'eval.o' depends on the file 'eval.c' and to construct 'eval.o' you must execute the command

```
cc eval
```
where cc is a C compiler command that compiles a '.c' file to form a '.o' file.

The final two entries (entry 7 and entry 8) in the makefile describe some extra dependencies for the '.o' files. For example, the last line states that, as well as depending on their respective '.c' files, the three '.o' files (main, eval and calc) also depend on the two header files 'stdio.h' and 'calc.h', therefore, if either one of these header files is changed it will be necessary to reconstruct 'main.o', 'eval.o' and 'calc.o' by applying the C compiler to their respective sources.

Implicit Rules

Implicit rules allow you to specify general dependencies between files of different types and the method of producing one type from another.

They are used by Make when the following conditions are met:

- A file which has to be recreated has no command sequence specified in an explicit rule in the makefile.
- One of the implicit rules in the makefile has a target type (either suffix or prefix) that matches the files type.
- The source file, whose name is constructed using the rule, exists.

Then, and only then, will the command sequence associated with that implicit rule be used. The source file becomes a prerequisite of the file.

An implicit rule consists of a source type, a target type and a command sequence. The source and target types must be joined together in that order and followed by a colon. The command sequence then follows in the same format as for explicit rules. For example:

```
.c.o :; echo "compiling $*"
   cc $*
```

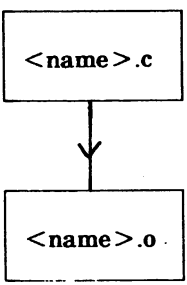
or

```
.c.o :;
   echo "compiling $*"
   cc $*
```

or

```
.c.o : [
   echo "compiling $*"
   cc $* ]
```

are all valid ways of writing the same implicit rule, the meaning of which is: to get from '.c' files to '.o' files, execute the commands 'echo "compiling $*"' and 'cc $*'. A diagram to show this is as follows:



where <name> can be any valid filename body. The $* in the command is a macro, the full meaning of which will be explained later. For now it is enough to say that it expands into the filename to be compiled.

A file's 'type' is usually determined by the letters following the period in its name. In effect what we are doing is classifying files by their suffix. Another way to classify files is by the directory in which they are in, or in other words by their prefix. The examples you have seen so far have all been suffix implicit rules. An entry for a prefix implicit rule has the same format as a suffix implicit rule. For example:

```
asm/o/ :; assem asm/$* to o/$*
```

is a prefix implicit rule for assembling assembler files in a directory called "asm" and producing object files in a directory called "o".

Having both suffix and prefix implicit rules gives you more scope in the way you can arrange your files. If you prefer to have different types of files in different directories then you should use prefix implicit rules. If on the other hand, you prefer all your files to be in the same directory and classified by their type then you should use suffix implicit rules. Any combination of the two forms is, of course, possible.

This method of dealing with prefixes means that files are limited to being just one directory level below the level in which Make was invoked.

The following example is an improved makefile for the program 'calc'.

```
#
# Example makefile - second attempt showing the
#                    use of implicit rules
#

# The implicit rules (prefix)

.asm.o :; assem $*.asm to $*.o
.c.o  :; cc $*

# The explicit rules

calc : go.o io.o main.o eval.o calc.o
   alink "go.o io.o main.o eval.o calc.o" to calc

go.o io.o : calc.i
main.o eval.o calc.o : stdio.h calc.h
```

Note that all suffix implicit rules are searched before any prefix implicit rule.

Macros

Make supports a macro facility that allows substitution of a sequence of instructions for a macro name. For those of you who have not met macros before, a macro is usually used to reduce the amount of text and therefore the amount of typing. It also aids clarity by, for instance, associating meaningful names with lists of files. Wherever a macro is used it expands into the text it was associated with when it was defined. Obvious candidates for macros are pieces of text that are used more than once in a file.

A macro is defined by an entry in the makefile consisting of the macro name, followed by an equals sign, followed by the text of the macro. The text may optionally be enclosed in square brackets in the same manner as for rules. For example:

```
FILES = first.asm second.asm last.asm
```

or

```
FILES = [ first.asm second.asm last.asm ]
```

both define the macro 'FILES' to be the text 'first.asm second.asm last.asm'. This macro can now be used in the remainder of the makefile. To do so use the macro name enclosed in angle brackets or parentheses directly preceded by a dollar. For instance:

```
$(FILES) : ASMHDR
```

or

```
$<FILES> : ASMHDR
```

are both ways of saying that each of the three files 'first.asm', 'second.asm' and 'last.asm' depends on the file ASMHDR.

Special Macros

Make predefines the meaning of certain special macros. It will become evident from their definitions that the text associated with these special macros depends on the rule they are being used in. There are four special macros. They are $@, $*, $<, and $n (where n is a alphanumeric character). They have the following meanings:

$@ The name of the target file (the file actually being recreated).

$n The nth prerequisite file specified in the explicit rule from where the command sequence was taken. n is a character in the range 1-9 or a-z to facilitate referencing up to 35 prerequisite files. For example:

```
graph.lib : draw.o plot.o paint.o
   join $1 $2 $3 AS $@
```

When the above rule is used to recreate graph.lib, the command line passed to join, after all the macros have been expanded, is as follows

```
draw.o plot.o paint.o AS graph.lib
```

join is a program that concatenates a list of files to one file.

$* The part of the filename shared by the target and the prerequisite in implicit rules. For example:

```
.c.o :; cc -n $*
```

When the above rule is used to recreate the target file 'ace.o', for instance, the command line passed to cc after the macro $* have been expanded is as follows:

```
-n ace
```

$< The name of the prerequisite file that caused the action in implicit rules. For example:

```
.b.o :; bcpl $< to $@
```

When the above rule is used to recreate the target file "blib.o", the command line passed to bcpl after the macros have been expanded is as follows:

```
blib.b to blib.o
```

A single dollar character is represented by two dollar characters within a makefile. Here is our example again, this time using macros.

```
#
# Example makefile - third attempt showing
#                    the use of macros
#

# The files

CFILES   = main.c eval.c calc.c
ASMFILES = go.asm io.asm
OFILES   = go.o io.o main.o \
           eval.o calc.o
```

```
# The implicit rules

.asm.o :; assem $*.asm  to  $*.o
.c.o  :; cc $*

# The explicit rules

calc : $(OFILES)
   alink "$(OFILES)" to calc

$(ASMFILES) : calc.i
$(CFILES)   : stdio.h calc.h
```

Directives

There are several directives which may appear as an entry on a single line anywhere in the makefile. These are direct instructions to Make. Note that angled brackets are used only for the sake of clarity. These brackets should not be typed explicitly as part of the directive; they just serve to enclose what sort of thing you should type. Directives available are as follows:

.START : <command seqence>
defines a sequence of commands to be executed before all others. If Make has no commands to execute in order to bring files up-to-date, then these commands are not executed. The command sequence is in the same format as for a rule entry.

.END : <command sequence>
defines a sequence of commands to be executed after all others. These commands are only executed if Make has had to do something.

.SILENT :
instructs Make to execute all commands without reporting any of them to the terminal.

.SUFFIXES : <list of suffixes>
informs Make of the order in which suffix implicit rules should be searched. The default action is to search in the order specified in the makefile.

.PREFIXES : <list of prefixes>
informs Make of the order in which prefix implicit rules should be searched. The default action is to search in the order specified in the makefile.

.INCLUDE : <file>
instructs Make to expand <file> into the makefile in place of the .INCLUDE line.

```
#
# Example makefile - fourth attempt
#

# The files

CFILES   = main.c eval.c calc.c
ASMFILES = go.asm io.asm
OFILES   = go.o io.o main.o \
             eval.o calc.o

# The implicit rules this time read from
# the standard file.

.INCLUDE :; make:make.inc

# Another directive

.SILENT :;

# The explicit rules

calc : $(OFILES)
   alink "$(OFILES)" to calc

$(ASMFILES)  : calc.i
$(CFILES)    : stdio.h calc.h
```

The backslash is used in this example as a line continuation character. (See the section on "Invoking Make" for more information on line continuation.)

Makefile Format

Filenames

Make allows filenames to contain any printable characters except semicolon, open square bracket, and equals The length of a filename is limited to 80 characters.

Comments

A comment is introduced by a '#' or bar '|' character. All text after and including the hash or bar on the same line is ignored by Make. For example:

```
#
# This is an example
#

target : prog1 prog2    # this is a comment
```

Line Continuation

A backslash '\' at the end of a line causes Make to treat the next line as part of the same line. This allows lists of filenames to be set out on more than one line of the makefile. A backslash anywhere else in a makefile has no special significance. For example:

```
TARGETS = type.obj dir.obj list.obj \
          copy.obj rename.obj edit.obj \
          delete.obj print.obj
```

This sets up a macro TARGETS to be equivalent to the eight filenames specified in the list above.

Executing Commands

The command lines for each command Make has to execute are first written to a temporary file. When Make has finished processing the makefile and has identified all the commands to be executed, the temporary file is executed as a batch file. The command line of each command be to executed is reported to the terminal at the same time as it is written to the temporary file. This reporting can be suppressed either using the command line switch '-s' or the makefile directive '.SILENT'.

Invoking Make

Make is a CLI command, and accepts the following template:

```
MAKE "TARGET -F, -T/S, -N/sS, -S/S"
```

where the TARGET parameter specifies the name of the primary target file. If this is omitted then the first target to appear in the makefile is used. The meanings of the other parameters are as follows:

-F File. Introduces the name of the makefile. If this is omitted then the name 'makefile' is assumed.

-T Touch targets. Informs Make to update the dates of the targets, without executing any commands. (See "TOUCH" utility.)

-N No execution. Informs Make not to execute the commands needed to update the targets, but just report the commands that would have been executed.

-S Silent execution. Informs Make not to report commands to be executed to the terminal.

Differences from UNIX Make

For those of you who are used to using UNIX Make, here is a list of the differences between UNIX Make and Metacomco Make:

- Unlike the UNIX version, Make has no built-in implicit rules. Instead a standard makefile is supplied which can be included into your own makefile using the .INCLUDE directive.

- Because filenames may contain colons (unlike under UNIX), the colon character when used as a separator must be separated from any filename by a blank.

- The use of the tab character in makefiles is not essential for Metacomco Make.

- Metacomco Make allows macro text and sequences of commands to be enclosed in square brackets, allowing macros to go over more than one line and commands to be specified in a freer format.

- Metacomco Make includes prefix implicit rules.

- Not all the directives available under UNIX Make are available from Metacomco Make. Those not available are .DEFAULT, and .PRECIOUS.

- Metacomco Make includes the directives .START and .END.

- Special macros $? and $% are not supported by Metacomco Make.

- The need for the .SUFFIXES directive is not essential under Metacomco Make.

- The '::' form of explicit rules is not supported by Metacomco Make.

Makefile Examples

To aid your understanding of makefiles, this section presents some examples.

Example 1

The first example is a makefile for maintaining a Metacomco product called Menu+.

```
#
# makefile for MENU+
#

# The files involved

CSOURCES = menu.c glob.c history.c objects.c \
           parse.c sdate.c errors.c
OBJECTS  = menu.o glob.o \
           history.o objects.o \
           parse.o sdate.o \
           errors.o tinyup.o

# The rules (alternatively the standard makefile
# could have been included)

.SUFFIXES : .c .asm .o

.c.o    :; cc $*

.asm.o :; assem $*.asm to $*.o

# The dependencies

menu+ : $(OBJECTS) ;
    alink "$(OBJECTS)" to menu+

$(CSOURCES)  :  menu.h
```

Example 2

This example maintains a system of commands. Each command, apart from 'help', is in a single source file and has been written in either Pascal, C, or assembler. These commands are handled by implicit rules. The 'help' command is handled separately. Note the use of the dummy target 'commands' to ensure all the commands are made.

```
#
# makefile for imaginary system
#

# The files

HELPFILES = help1.c help2.c help3.c help4.c
PASFILES  = pete.pas sheila.pas
CFILES    = andy.c wendy.c alan.c \
            liz.c tim.c nick.c \
            $(HELPFILES)
ASMFILES  = paul.asm margaret.asm \
            chris.asm dennis.asm

COMMANDS  = help.obj \
            pete.obj sheila.obj \
            andy.obj wendy.obj alan.obj liz.obj \
            paul.obj margaret.obj \
            chris.obj dennis.obj

# A useful command

COPY      = copy $1 to $@

# Some directives

.END    :; echo "That's all folks"
.IGNORE :;

# The rules

.pas.obj : [
```

```
PASCAL $*.pas to $*.o
alink$*. to $*.obj
]

.c.obj  : [
cc $*
alink $*.o to $*.obj
]

.asm.obj : [
assem $*.asm to $*.o
LINK $*.o to $*.obj
]

# the explicit rules

commands : $(COMMANDS)

$(ASMFILES) : comhdr
asmhdr : sys:\com\comhdr ; $(COPY)

$(CFILES)  : coms.h

$(PASFILES) : com.inc

# help is more difficult

help.obj : $(HELPFILES)

 alink "$(HELPFILES)" to help.obj
$(HELPFILES) : help.h
```

MAKE Error Messages

General errors

`Bad arguments`

Make was supplied with an incorrect parameter line.

`Cannot open T:make-commands-nn for output`

Make failed to open its workfile, possibly because T: is not assigned.

`Cannot open makefile <file>`

Make failed to open the makefile, possibly because it does not exist.

`Cannot get any more workspace`

Make has run out of memory for its workspace.

`No target`

No primary target file was specified in either the makefile or the parameter line.

`Circularity detected`

The dependencies specified in the makefile result in a file depending on itself. This is not permitted.

`<file> is up-to-date`

The primary target file is up-to-date. Make has nothing to do.

`No commands to issue`

All targets are up-to-date.

Do not know how to make <file>

There are no commands specified in the makefile to recreate a file that make must recreate.

Errors in Makefile

Unexpected separator

A separator character, either a colon, semicolon, left square bracket, newline, or equals, is misplaced in the makefile.

Illegal separator

A colon, semicolon, or equals was expected.

Target file name too long

File name longer than 80 characters.

Prerequisite file name too long

File name longer than 80 characters.

Spurious character found

Illegal character in makefile, probably a non-alphanumeric character in a file name.

Unexpected end of file

The makefile ends in the middle of a command sequence. A right square bracket is expected.

More than one command sequence for file

A file is a target in more than one explicit rule which has a command sequence associated with it.

No command sequence for implicit rule

An implicit rule has no commands specified in its definition.

Command line too long

Command line longer than 255 characters. This is the limit for a single command line.

Unknown directive

A directive name has been misspelt or an attempt has been made to use a directive that is not supported.

.INCLUDE file name too long

File name longer than 80 characters.

Cannot open .INCLUDE file <file>

Make failed to open the file specified in a .INCLUDE directive.

Suffix too long

One of the suffixes specified in a .SUFFIX directive is longer than 80 characters.

Illegal suffix

One of the suffixes specified in a .SUFFIX directive does not begin with a dot, '.', character.

Prefix too long

One of the prefixes specified in a .PREFIX directive is longer than 80 characters.

Illegal prefix

One of the prefixes specified in a .PREFIX directive does not begin with a backslash, '\', character.

More than one .START command sequence

Only one .START command sequence is permitted in a makefile.

More than one .END command sequence

Only one .END command sequence is permitted in a makefile.

Macro Errors

Incorrect macro definition

More than one name is specified before the equals separator.

Macro name too long

Macro name longer than 80 characters.

Undefined macro

A macro is used before it has been defined.

Program Execution Errors

<file> failed - error code <n>

Either Make was unable to load the program <file> or the program loaded

MOUNT

Format: MOUNT <device>

Template: MOUNT "DEV/A"

Purpose: To mount a new device.

Specification:

The MOUNT command mounts (makes available) a non-standard device. Standard devices (sys:, ser:, etc) are mounted automatically by the system.

To use MOUNT, type MOUNT followed by the name of the device (for example, aux:). MOUNT then reads the file DEVS:MOUNTLIST, finds the description for that device, and makes it available for use. The device is not actually initialized, however, until it is first referenced.

Examples:

```
MOUNT pipe:
```

mounts the pipe handler device 'pipe:'.

See also: PIPES, AUX-CLI.

PACK

Format: PACK [FROM] <filename> [TO] <filename> [[WORKSIZE | WS] <n>]

Template: PACK "FROM/A,TO/A,WORKSIZE=WS"

Purpose: To translate a text file to a compacted form.

Specification:

PACK helps you save space on a disk. It can compress a file into a compacted form saving as much as 50% of the original space.

The FROM parameter specifies the text file to be translated. The TO parameter specifies the output file that will contain the compacted version. The WORKSIZE parameter specifies the worksize in bytes. The default worksize is 10000 bytes.

There is an overhead of 1000 bytes associated with each compacted file. The original text file must be at least 2000 bytes in size before any reduction in size occurs.

PACK will work on a non-text file, but the reduction is not so good.

See also: UNPACK

PIPES

Format: PIPE:[<string>]

Purpose: To communicate I/O between tasks.

Specification:

A pipe is a mechanism whereby output from one task can be transmitted as input for another task. The device pipe: handles all pipes. A pipe may have a name of the form 'pipe:<string>' (similar to a file at the top level of the filing system). Pipes only exist while data is being transferred; as soon as all the data has been read the pipe is terminated. Data can only be transferred sequentially, no seeking may take place.

Note that this communication is only possible between two tasks, so at least one of the tasks using a pipe will have to be RUN or set going from a second CLI. The task reading from the pipe will only have its read requests satisfied when the task writing to the pipe has written some data. Before pipes can be used it is first necessary to mount the device pipe: (see MOUNT earlier in this manual for further details).

Examples:

```
RUN COPY myfile to pipe:mypipe
TYPE pipe:mypipe
```

types the contents of the file 'myfile'. The file is transferred by way of the pipe 'pipe:mypipe'.

```
RUN DISASM myfile.o to pipe:t
BROWSE pipe:t
```

browses the disassembly listing of the file 'myfile.o'.

TOUCH

Format: TOUCH [FILE] <filename>

Template: TOUCH "FILE/A"

Purpose: To update a file's creation date.

Specification:

This command is used in conjunction with MAKE. Sometimes it is necessary to fool Make into believing it has nothing to do. You may have changed a source file but realised that it is unnecessary to recreate the object. To stop Make recreating the object file it is necessary to change the creation time of the object file to the current time. Touch is a utility that updates the time of a file without altering the contents. Touch accepts the following template:

```
TOUCH "FILE/A"
```

The single parameter is a file specification that may include the standard pattern matching characters. Touch updates the times of all the files that fit the specified pattern. For example,

```
TOUCH #?.o
```

up-dates the time of all the '.o' files in the current directory.

UNPACK

Format: UNPACK [FROM] <filename> [TO] <filename> [[WORKSIZE | WS] <n>]

Template: UNPACK "FROM/A,TO/A,WORKSIZE=WS"

Purpose: To regenerate a text file from a compacted form.

Specification:

The FROM parameter specifies the compacted file to be regenerated. The TO parameter specifies the output file that will contain the regenerated text file. The WORKSIZE parameter specifies the worksize in bytes. The default worksize is 10K bytes.

Examples:

```
PACK myfile.text to myfile.pack
```

generates a compacted version of the text file 'myfile.text' as the file 'myfile.pack'.

```
UNPACK myfile.pack to myfile.text
```

regenerates the text file 'myfile.text' from the compacted file 'myfile.pack'.

See also: PACK

Appendix: Customer Support

Customer support will be given by Metacomco to all registered users. If you have not sent us your registration card, please do so now!

Technical Support

In order that we can maintain our high standard of customer support, we would ask you to please follow these guidelines.

1 Written queries can be by letter, electronic mail or facsimile. Electronic mail can be sent on Dialcom or Telecom Gold where our mail address is System 84, mailbox number MEA001. Our facsimile number is (Group 3) (0272) 428618.

2 You can also leave mail on our BIX (BYTE Information Exchange) mailbox. Our conference name on this system is 'metacomco'.

3 It will nearly always be more efficient to send your questions by one of the above methods rather than telephoning us. Priority will be given to the above methods.

4 For all technical enquiries please include the following details:

- your name and full mailing address (or mailbox details)
- your user registration number
- the version number of the software
- the version number of the operating system
- the hardware configuration you are using (for example, RAM disk, disk drives, etc)
- details of any additional software you are using.

5 If you appear to have found a bug try to create a small program that reproduces the problem. Send a listing, if no longer than 20 lines, otherwise enclose a copy on disk. Remember to include listings of any #include files (and watch out for #include file nesting).

6 Always give as full a description as possible of the problems you have encountered. This will help us to help you.

7 A faulty item, if purchased from Metacomco and still within the guarantee period, must be returned to us for replacement. If outside the guarantee period, we will still replace faulty items but will charge a small amount to cover our costs and postage. If you have purchased the software from a dealer rather than from Metacomco directly, then return it to them for replacement

8 We welcome any suggestions about our products but please make them in writing.

Upgrades

You will be informed, either individually or through advertisements, of all upgrades to our products. We may have to make a charge for an upgrade depending on the extent of the changes.

To get your upgrade please send the required amount to Metacomco and either include your product master disks or quote your registration number.

IMPORTANT

READ THIS CARD BEFORE USING THIS PRODUCT

1 You must study the terms and conditions of the software licence printed on the back of this card before starting to use this product. If you do use the materials contained in this package, Metacomco will assume that you have read this licence, understood it, and agreed to be bound by it. If you do **not** accept the terms of this licence, you should promptly return the unused package to the vendor from whom you purchased it, and your money will be refunded.

2 If you accept the licence, you should fill in the tear-off registration card attached to this, and post it. Registered users are entitled to the following technical support, in addition to the rights given by the licence.
Metacomco will reply to technical enquiries relating to the use of the software products included in this package. Enquiries must be **in writing**, quoting your user registration number, and must be accompanied by a stamped addressed envelope for the reply. No telephone enquiries can be handled. Non-registered users will not be supported.

Your User Registration Number

C 6649

METACOMCO

26 Portland Square, Bristol, BS2 8RZ

METACOMCO PROGRAM LICENCE

1 LICENCE AGREEMENT

METACOMCO has developed and owns rights in the Program, including the right to licence others to use it. You, the end-user, wish to be licenced to use this Program and therefore agree to the following terms and conditions.

2 LICENCE

METACOMCO grants you a non-exclusive licence to do the following:

a use the Program on a single computer system ("the licenced system");

b make copies of the Program for backup or modification purposes, provided that no more than four such copies exist at any one time and that such copies are only used on the licenced system;

c modify the Program, for use only on the licenced system;

d transfer the Program and this licence to another person or entity provided that the other party agrees to all the terms and conditions of this licence and that you transfer or destroy all copies and modifications made of the Program. You do not retain any right with respect to the transferred package.

Any other act involving reproduction or use of, or other dealing in the Program is prohibited.

3 TERM OF THE LICENCE

a You may terminate this licence at any time by returning to METACOMCO, or destroying, all copies of the Program, including modifications, and all associated documentation.

b METACOMCO may terminate this licence at any time if you fail to comply with any term or condition of this licence agreement. Upon notification of such termination, you agree to return to METACOMCO or destroy all copies of the Program, including modifications, and all associated documentation.

4 METACOMCO'S RIGHTS

You acknowledge that METACOMCO owns the copyrights and other proprietary rights in the Program and associated documentation. You agree to take all reasonable steps to protect METACOMCO's rights including, but not limited to, placing METACOMCO's copyright notice on all copies or modifications of the Program which you make. You also agree that you will not decompile the object code version of the Program provided to you, or use any other means to produce a source code version of the Program.

5 LIMITED WARRANTY

The Program and the enclosed instructional and reference material are provided "as is" without warranty as to their performance, merchantability, or fitness for any particular purpose, or other warranty of quality. You are solely responsible for the selection of this Program to achieve your intended results, and for the installation of, use of and results obtained from the Program.

However, to the original purchaser only, METACOMCO warrants the magnetic media on which the Program is recorded to be free from defects in material and faulty workmanship under normal use and service for a period of ninety (90) days from the date of your receipt of the product. If, during this 90-day period, a defect in the magnetic media should occur, the magnetic media may be returned to METACOMCO or the dealer from whom you purchased the Program, and METACOMCO will replace the magnetic media without charge to you, provided that you have previously sent in your registration card to METACOMCO.

THIS WARRANTY IS GIVEN IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL METACOMCO BE LIABLE FOR CONSEQUENTIAL DAMAGES EVEN IF METACOMCO HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

6 GENERAL

Upon termination of this Licence under paragraph 2(b) METACOMCO may also seek any legal or equitable remedy available against you for any violation of the terms of the licence.

If you are resident in, and the Program has been supplied in, the United States of America, it is agreed that the laws of California shall govern without reference to the place of execution or performance, except as to copyright matters which are covered by Federal laws. If you are resident, and the Program has been supplied, elsewhere other than in the United States of America, it is agreed that the laws of England shall govern without reference to the place of execution or performance.

No statements in this licence shall affect the statutory rights of consumers.

METACOMCO

26 Portland Square, Bristol BS2 8RZ, UK